AFRL-RI-RS-TR-2008-156
In House Interim Technical Report
June 2008

MULTI-AGENT PLANNING IN DYANMIC DOMAINS

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

NOTICE AND SIGNATURE PAGE

AFRL-RI-RS-TR-2008-156 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/s/	/s/
ALEX F. SISTI	JAMES W. CUSACK
Chief, Information Systems Research Branch	Chief, Information Systems Division
	Information Directorate

REPORT DOCUMENTATION PAGE	*Form Approved* **OMB No. 0704-0188**

1. REPORT DATE *(DD-MM-YYYY)* Jun 2008	**2. REPORT TYPE** Interim	**3. DATES COVERED** *(From - To)* Apr 06 – Mar 08

4. TITLE AND SUBTITLE MULTI-AGENT PLANNING IN DYNAMIC DOMAINS	**5a. CONTRACT NUMBER** In House
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER 62702F
6. AUTHOR(S) James H. Lawton, Matthew Berger	**5d. PROJECT NUMBER** 558S
	5e. TASK NUMBER HA
	5f. WORK UNIT NUMBER TR
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) AFRL/RISB, 525 Brooks Rd, Rome NY 13441-4505	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) AFRL/RISB, 525 Brooks Rd, Rome NY 13441-4505	**10. SPONSOR/MONITOR'S ACRONYM(S)**
	11. SPONSORING/MONITORING AGENCY REPORT NUMBER AFRL-RI-RS-TR-2008-156

12. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED. PA# WPAFB 08-3399

13. SUPPLEMENTARY NOTES

14. ABSTRACT
The issue of multi-agent planning in highly dynamic environments is a major impediment to conventional planning solutions. Plan repair and replanning solutions alike have difficulty adapting to frequently changing environment states. To adequately handle such situations, this paper instead focuses on preserving individual agent plans through multi-agent coordination techniques. We detail a reactive agent system architecture in which the main focus of an agent is to be able to achieve its subgoals without interfering with any other agent. The system is a 3-level architecture, where each level is guided by the following fundamental principles, respectively: *when* is it valid to generate a plan for a subgoal, *who* is most appropriate for completing the subgoal, and *how* should the plan be carried out.

15. SUBJECT TERMS
Distributed Planning, Dynamic Reasoning, Intelligent Agents

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** James H. Lawton
a. REPORT U	**b. ABSTRACT** U	**c. THIS PAGE** U	UU	28	**19b. TELEPHONE NUMBER** (*Include area code*) N/A

Contents

List of Figures

1 Executive Summary

Intelligent software agents will almost certainly be a critical component of future Air Force (AF) distributed, heterogeneous information and decision-support systems. The Agents Technology Research Group of the Air Force Research Laboratory's Information Directorate is committed to ensuring that the core technologies needed to support agent-based systems for AF needs are being developed. One of these core technologies is the area of distributed multi-agent planning in highly dynamic domains, a key component of the AF future Command and Control strategy [2]. Multi-agent planning in the context of rapidly-changing environments, however, becomes an involved, unmanageable problem rather quickly. Further, maximizing concurrency and ensuring conflict-free plans becomes a difficult problem to manage when using techniques like plan repair and replanning. Therefore, multi-agent coordination through conflict resolution of individual agent plans can be an effective way to ensure safe planning while providing a reasonable execution time.

This report describes an approach that rapidly merges individual agent plans in highly dynamic environments. Agents generate their own plans separately, and use a coordination process that is concerned with maximizing parallelism and minimizing conflicts in the execution of the plans. The algorithm is targeted towards environments in which a global goal may be difficult for a single agent to achieve alone, but becomes more manageable when the goal can be partitioned into subgoals that can be allocated to several cooperating agents. As each agent generates a plan for just its own subgoals, the problem reduces to agent coordination, by ensuring that no conflicts exist between plans.

To accomplish this, we have developed and implemented a 3-level multi-agent coordination architecture, where each level resolves fundamental issues in agent conflicts. The first level handles the issue of *when* it is valid to plan for a subgoal. The second level addresses the issue of *who* should be made responsible for achieving a subgoal. The third level deals with the issue of *how* to achieve a subgoal, in which agents reserve resources for planning out primitive actions.

To ground this discussion we use the domain of Sokoban as a running example to better illustrate the system. Sokoban is a simple discretized box pushing environment in which a mover must push all boxes in the environment to goal squares. Despite its simple premise, the game becomes extremely difficult to manage as the number of boxes and the size of the environment increase. It has in fact been shown that the game is PSPACE-Complete [5]. We have extended the basic Sokoban environment to include multiple movers, each represented by a planning agent. *Multi-agent Sokoban* is representative of a class of environments in which the separate agent plans are inherently disjoint. It also can be viewed as an environment in which space is a highly contentious resource, requiring careful agent coordination to ensure that the agent plans are conflict free.

2 Introduction

Intelligent software agents will almost certainly be a critical component of future Air Force (AF) distributed, heterogeneous information and decision-support systems. The Agents Technology Research Group of the Air Force Research Laboratory's Information Directorate is committed to ensuring that the core technologies needed to support agent-based systems for AF needs are being developed. One of these core technologies is the area of distributed multi-agent planning in highly dynamic domains, a key component of the AF future Command and Control strategy [2]. Multi-agent planning in the context of rapidly-changing environments, however, becomes an involved, unmanageable problem rather quickly. Further, maximizing concurrency and ensuring conflict-free agent plans becomes a difficult problem to manage when using techniques like plan repair and replanning. Therefore, multi-agent coordination through conflict resolution of individual agent plans can be an effective way to ensure safe planning while providing a reasonable execution time.

This report describes an approach that rapidly merges individual agent plans in highly dynamic environments. Agents generate their own plans separately, and use a coordination process that is concerned with maximizing parallelism and minimizing conflicts in the execution of the plans. The algorithm is targeted towards environments in which a global goal G may be difficult for a single agent to achieve alone, but becomes more manageable when the goal can be partitioned into subgoals $G = S_1 \wedge S_2 \wedge ... \wedge S_n$, in which we have n total agents, and each agent A_i is responsible for goal S_i. Correspondingly, each agent A_i generates a plan P_i. The problem now reduces to agent coordination, by ensuring that no conflicts exist between plans.

For environments in which resources have high levels of contention, it is unlikely that each agent's plan can be executed without conflicting with some other agent's plan. More specifically, an agent's list of primitive actions is likely to run into problems with another agent's actions. Thus, we propose a system architecture capable of dealing with such environments, consisting of two basic principles: *goal abstraction* and *flexible agent behavior*.

Goal abstraction refers to decomposing a plan into a hierarchy of subgoals to be achieved by the plan. In contentious environments, coordinating separate agent subgoals is far more effective than relying on primitive actions, as enforcing synchronization actions on subgoals ensures that plans expanded on these subgoals will be conflict free [4, 9].

Flexible agent behavior refers to agents not being bound to their initial plans. An agent bound to its plan in a contentious environment will find it difficult to replan and/or use plan-repair as time progresses [7]. The combination of goal abstraction and flexible agent behavior is well suited for highly dynamic environments: as subgoal conflicts are worked out, agents are aware of subgoals to be achieved, even if they initially belong to another agent, and plan from there. Note the distinc-

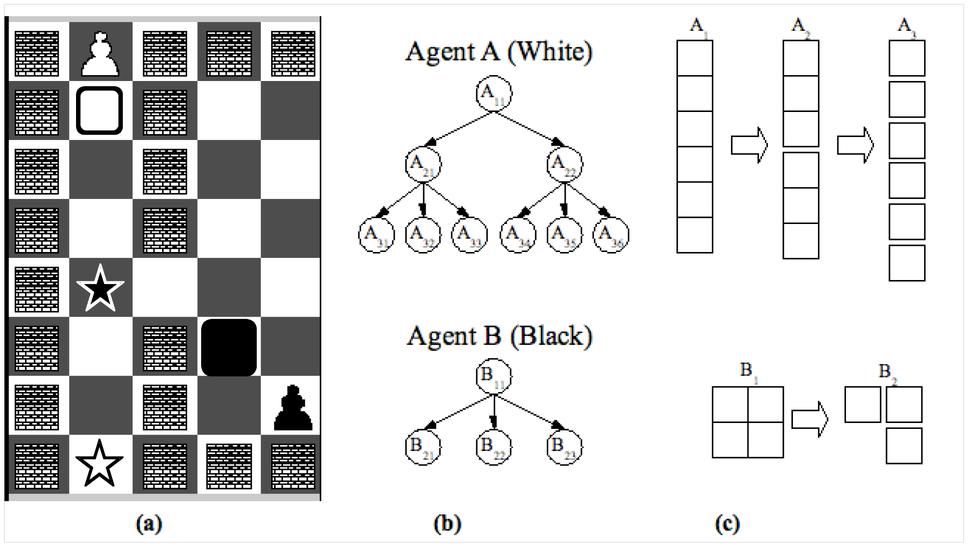

Figure 2.1: An example Sokoban scenario. (a) The white agent must push the white box all the way down, while the black agent must push the box up, then push it to the left. (b) The resulting subgoal hierarchies for both agents. (c) Resource usage for all subgoals.

tion between subgoals and partial plans, where partial plans commonly characterize hierarchical planning. It is implicit that when a partial plan is refined, it still belongs to the agent responsible for it. An agent may own a subgoal, but is not necessarily bound to the plan that results in the subgoal.

Motivated by these observations we have developed a 3-level multi-agent coordination architecture, where each level resolves fundamental issues in agent conflicts. These issues are: *when* is it valid to generate a plan for a subgoal, *who* is most appropriate for completing the subgoal, and *how* should the plan be carried out. This architecture is presented in detail in Chapter 3.

2.1 Multi-agent Sokoban

To ground this discussion we use the domain of Sokoban as a running example to better illustrate the system. Sokoban is a simple discretized box pushing environment in which a mover (or multiple movers) must push all boxes in the environment to goal squares. The mover must be “behind” a box to push it in a given direction (i.e., on the opposite side of the box from the desired destination). Walls and other boxes in the environment serve as impediments that must be navigated around. Despite its simple premise, the game becomes extremely difficult to manage as the number of boxes and the size of the environment increase. It has in fact been shown that the game is PSPACE-Complete [5].

Sokoban in the context of a single agent becomes an unmanageable problem rather quickly, therefore motivating the desire to split the problem up among multiple agents. If we assign a subset of boxes to each agent, in which each agent is responsible for producing a plan that solves its separate goals, the problem now becomes more manageable as we are significantly limiting the search space for each agent. Our main objective now becomes ensuring each agent may execute its plan conflict free with regard to the other agents.

Multi-agent Sokoban is representative of a class of environments in which the separate agent plans are inherently disjoint. It also can be viewed as an environment in which space is a highly contentious resource, requiring careful agent coordination to ensure that the agent plans are conflict free. In this case, an agent's plan will likely have a number of conflicts with other agents as time progresses. Each agent thus should not be committed to its original plan and should be more concerned with, at the very least, ensuring their subgoals are completed in a safe manner and do not conflict with other agent subgoals.

3 Methods, Assumptions and Procedures

As introduced in Chapter 2, we have developed a 3-level multi-agent coordination architecture, where each level resolves fundamental issues in agent conflicts. The first level, detailed in Section 3.1, handles the issue of *when* it is valid to plan for a subgoal. The second level, detailed in Section 3.2, handles the issue of *who* should be made responsible for achieving a subgoal. The third level, detailed in Section 3.3, handles the issue of *how* to achieve a subgoal, in which agents reserve resources for planning out primitive actions. Figure 3.1 presents the complete general algorithm representing the architecture.

3.1 Coordinating Temporal Consistency

Verifying that subgoals are *temporally consistent* with respect to one another refers to the issue of *when* is it valid to plan for a subgoal. To start, assume each agent A_i has generated a plan P_i to achieve its goal S_i, where their are n agents and the global goal G is decomposed into separate goals for each agent, such that $G = S_1 \wedge S_2 \wedge ... \wedge S_n$. Furthermore, assume a goal S_i can be decomposed into a hierarchy of subgoals where at a particular depth of the hierarchy, there exists a set of subgoals that fully compose the original goal. This representation is very much in line with hierarchical task networks [6] and hierarchical plans [4]. Furthermore, each subgoal must be defined in such a way that the resources it encompasses must be expressed in a compact manner, yet never understating the resource usage. This is similar to summary conditions in [4], except that in [4] summary conditions consist of STRIPS-like syntax of aggregated preconditions and effects, while we choose to represent resource usage. The subgoal depth represents how fine-grained each subgoal is in terms of resource usage.

Figure 2.1 shows an example of subgoal hierarchies for a simple Sokoban scenario. Our implementation of a Sokoban game server is based the Asynchronous Chess server [8], modified for the Sokoban domain to support synchronized actions. The pawns represent the respective movers which belong to each agent. The white pawn belongs to agent A and the black pawn belongs to agent B. Each agent generates its own separate plan for achieving its assigned goals (e.g., boxes to be moved to goal locations). The stars ($\star$) represent the goal locations for each agent, and each agent is assigned a set of boxes (e.g., the white agent plans for the white box to be pushed into the white goal). In constructing the subgoal hierarchy, we view a single box push as being the lowest level subgoal, corresponding to a primitive "movement" action. More abstract goals are represented as being a series of box pushes, or in terms of resources, the full amount of area in

```
function main (subgoal queue $Q$, empty plan $P$, global goal $G$)
while $G$ not completed do
  // Level 1: WHEN is a subgoal valid?
  ensure temporal validity of next $S \in Q$
  obtain leaf subgoals $\{q_1, q_2, ..., q_n\} \subseteq S$
  while qualitative time not met do
    // Level 2: WHO is assigned each subgoal
    auction each subgoal $q \in S$
    engage in subgoal auction with all other agents
    obtain won subgoals $R$
    // Level 3: HOW is each subgoal planned for
    for next $r \in R$ ensure resource exclusion
    plan for $r$, add to $P$
  end while
end while
```

Figure 3.1: Algorithm detailing the 3-level system architecture.

which boxes are pushed. For instance, agent A's second level hierarchy depth indicates that each subgoal is essentially the action of pushing a box 3 times. Note that resource reservation at every level of box pushes only represents where the boxes are, but not the final box push. In other words, it is roughly the summation of preconditions, but expressed in a more compact manner in terms of resources.

3.1.1 Temporal Constraint Matrices

Assume that an agent A has a hierarchy of subgoals $G = (G_1, G_2, ..., G_m)$, where m represents the depth of the hierarchy, and each element, G_i, is itself a set of subgoals for that particular depth. Also assume an agent B has its own hierarchy of subgoals $H = (H_1, H_2, ..., H_n)$, where n represents the depth of the hierarchy, and each element, H_j is a set of subgoals for that particular depth. We may represent the temporal constraints of B imposed on A by representing the conflicts that exist between resource usage in their subgoals.

Now let us assume that we choose an arbitrary level in the goal hierarchy for A, i, such that we have a list of (ordered) subgoals in G_i, as well as for B, j, where we have a list of (ordered) subgoals in H_j. Temporal conflicts caused by B on A may be represented by viewing the ordered subgoals as containing qualitative time for that particular hierarchy depth. For example, assuming the sets $G = (g_1, g_2, ..., g_k)$ and $H = (h_1, h_2, ..., h_l)$, the qualitative time of subgoal g_i is i. For each subgoal in G_i there exist potential conflicts across all subgoals in H_j, or in other words, at each qualitative time in G_i there exist potential conflicts across all qualitative times in H_j. This information can better be represented in terms of a matrix, which we call a *temporal constraint matrix*, where each row represents the agent attempting to achieve its subgoals, each column represents the opposing agent who may be in conflict, and each value in the matrix is a 0 or 1 indicating no conflict or conflict, respectively.

To ground this discussion, the following matrices represent conflicts between the two agents in Figure 2.1, using each agent's lowest subgoal hierarchy level (the third level for agent A and the second level for agent B):

$$M_{A\to B} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \quad M_{B\to A} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

The first matrix, $M_{A\to B}$, represents agent A's conflicts with agent B; similarly, the second matrix, $M_{B\to A}$, represents agent B's conflicts with agent A. As A's subgoals get processed, we are moving down its matrix, while at the same time moving right in B's matrix. Observe that when B is at its qualitative time $< 4 >$ (i.e., fourth column in $M_{B\to A}$), agent A runs into conflicts at qualitative time $< 3, 5 >$ (i.e., third through fifth rows of $M_{A\to B}$). That is, since in the example domain conflicts are spaces on the board, if B were to blindly follow its plan, it would move the black box directly into A's path, causing a conflict with A's plan to move the white box. Similarly, when A is at its qualitative time $< 4 >$, agent B has a conflict when it is at its qualitative time $< 3 >$. Our objective is to avoid these conflicts as agent subgoals get processed.

3.1.2 Deadlock Graphs

The temporal constraint matrices allow agents to reason only so much about long term temporal conflicts among agent subgoals. Therefore, we may also look at the matrix as in fact being a directed acyclic graph detailing the various states an agent A is currently in, regarding another agent B's subgoals. In the current form of the temporal constraint matrix, each entry can be considered a node that represents a conflict or lack of conflict. The graph is traversed by one of two ways: A planning for a subgoal reduces to moving down, while B planning for a subgoal reduces to moving right. Graph traversal fits in with how A represents rows in the temporal constraint matrix, and B columns.

Using this representation we may also identify nodes in the graph that will in fact lead to conflicts, despite these particular nodes not actually containing conflicts. First, the temporal constraint matrix is transformed into a directed acyclic graph. Nodes leading to conflicts may be derived by observing that a node whose children all contain conflicts will also result in conflicts. The full algorithm is detailed in Figure 3.2. Assuming agent A contains k subgoals and B contains l subgoals for arbitrary hierarchy depths we find that the worst-case time complexity of this algorithm is $\mathbf{O}(k^2l^2)$. However, in practice this rarely occurs, as the algorithm will usually iterate over the graph in linear time, making average-case time $\mathbf{O}(nkl)$, where n is the greater of k and l.

But representing the nodes as containing a conflict and not containing a conflict is still insufficient in reasoning about long term temporal conflicts. Thus, we can annotate the nodes to contain

```
function conflictDerivation ($i$ subgoals, $j$ subgoals)
graphChange ← true
while graphChange do
    graphChange ← false
    for $r = 0$ to $i - 1$ do
        for $c = 0$ to $j - 1$ do
            n ← $node_{r,c}$
            bC ← n.bottomChild, rC ← n.rightChild
            if ¬n.conflictExists ∧ bC.conflictExists ∧ rC.conflictExists
            then
                n.conflictExists ← true
                graphChange ← true
            end if
        end for
    end for
end while
```

Figure 3.2: Algorithm for deriving conflicting states.

meaningful information about states relative to the global goal. We term these graphs as *deadlock graphs*. Each node can be one of three types: a *safe node*, a *solvable node*, and a *deadlock node*. A safe node indicates that we may safely plan for our subgoal at our qualitative time. A solvable node indicates we may not plan for our subgoal at our qualitative time, however we may wait for an agent (or agents) to plan for its subgoals before we proceed. A deadlock node indicates a state in which we will never be able to plan for a subgoal without running into a conflict. As shown in the next section, these graphs are used in preventing conflicts among subgoals.

We may also further annotate the nodes of a deadlock graph for ease of processing. For a safe node, we may record the minimum qualitative time it will take before we reach our goal state. For a solvable node, we may record the minimum time to wait before we may plan again (i.e., before we reach a safe node). Figure 3.3 contains the full deadlock graph construction algorithm. Using the same representation we find the worst-case time complexity of the algorithm to be $\mathbf{O}(kl^2)$.

3.1.3 Subgoal Coordination

We now illustrate the subgoal coordination algorithm, in addition to how to use deadlock graphs for determining temporal validity. Refer to Figure 3.4 for the full algorithm. At the start, each agent broadcasts its most abstract subgoal to all other agents. Each agent then observes if it is temporally valid to process their subgoal (described in Sections 3.1.3.1 and 3.1.3.2). If all subgoals are found to be temporally invalid, we then refine the subgoal hierarchy by having each agent broadcast its subgoals at the next level of its hierarchy. This process is repeated until we find a temporally valid set of subgoals among agents. Note that *all* subgoals need to be sent for each hierarchy level. This is necessary for correctly inferring temporal consistency—subgoals only local to a qualitative

```
**function** constructDeadlockGraph ($G_i$, $H_i$)
**for** $r = i$ to $0$ **do**
   **for** $c = j$ to $0$ **do**
      n ← $node_{r,c}$
      **if** n.isGoal **then**
         n.goalCost ← 0
         n.solvableCost ← 0
      **else**
         traverseForDeadlock(n)
      **end if**
   **end for**
**end for**


**function** traverseForDeadlock ($n$)
**if** ¬n.conflictExists **then**
   bC ← n.bottomChild
   rC ← n.rightChild
   **if** ¬bC.isDeadlocked **then**
      n ← safe
      n.goalCost ← bC.goalCost + 1
   **else if** rC.exists ∧ ¬rC.isDeadlocked **then**
      n ← solvable
      n.solvableCost ← rC.solvableCost + 1
      n.goalCost ← rC.goalCost + 1
   **else**
      n ← deadlocked
   **end if**
**else**
   rC ← traverse right until non-conflict is found
   **if** rC.exists **then**
      n ← solvable
      n.solvableCost ← nodesTraversed
      n.goalCost ← rC.goalCost + nodesTraversed
   **else**
      n ← deadlocked
   **end if**
**end if**
```

Figure 3.3: Deadlock Graph Construction.

```
**function** coordination (subgoal hierarchy $Q$)
$q \in root(Q)$
level $\leftarrow$ 1
solutionFound $\leftarrow$ false
subgoalsExhausted $\leftarrow$ false
**while** $\neg$solutionFound $\wedge$ $\neg$sugoalsExausted **do**
  gather all subgoals $q_j$ at $level$
  solutionFound $\leftarrow$ temporarilyValid($q_j$)
  **if** $\neg$solutionFound **then**
    subgoalsExhausted $\leftarrow$ a.areLeafNodes
    **if** $\neg$subgoalsExausted **then**
      $q \in children(q)$
      level $\leftarrow$ level + 1
    **end if**
  **end if**
**end while**
```

Figure 3.4: Subgoal coordination algorithm.

time are insufficient in correctly determining conflicts. For efficiency we may cache each agent's subgoal hierarchy as it is sent, eliminating the need to re-broadcast subgoal hierarchy levels in the future.

It is possible to perform a more exhaustive search as in [4], however our aim is to minimize communication overhead among agents. Expanding the subgoal search per subgoal hierarchy level gives us an efficient means of finding a solution at the expense of foregoing certain subgoal level combinations. Note, however, that we are not sacrificing the possibility of finding a solution. The lowest levels of all subgoal hierarchies will be checked last, where this represents the most refined subgoals. If a solution exists at any combination of higher levels, it is guaranteed to exist at the lowest levels. If we have n agents and m leaf subgoals per agent, the time-complexity of this algorithm reduces to $\mathbf{O}(n\ log(m))$, assuming each agent's subgoal tree is well-balanced.

This coordination algorithm has several advantages. First, for environments in which agent plans are inherently disjoint, the algorithm will likely terminate at a high level of the subgoal hierarchy, resulting in small communication overhead. Second, an agent's privacy is maintained to the extent possible. For mutually disjoint plans, only the most abstract plan information is sent to agents. Under environments in which temporal conflicts are rampant, the leaf subgoals are forced to be shared. However, this is a necessity in ensuring temporal conflicts are to be avoided in such environments.

As illustrated in the coordination algorithm, we still need some way of defining temporally valid subgoals. We may very well use a costly centralized solution in observing all possible subgoal interactions for all agents. However since we are assuming a distributed environment and we are looking for fast coordination, we have developed two approaches (discussed below) that use local information in attempting to find a global solution. Common to the approaches is the idea of traversing each agent's deadlock graphs, such that we never enter a deadlock state. We however make no guarantee of finding a solution if one exists.

3.1.3.1 Assuring Temporal Validity

In the first approach, agents first broadcast their local deadlock graph information to all other agents. This consists of three nodes: the passive node, representing the node for which the other agent plans and we do not; the active node, representing the node for which we plan and the other agent does not; and the both node, representing both of us planning concurrently. Formally, an agent A_i constructs a triple to send to every other agent A_j, $C_{i,j} = \langle a_{i,j}, p_{i,j}, b_{i,j} \rangle$, where $a_{i,j}$, $p_{i,j}$, and $b_{i,j}$ represent A_i's passive, active, and both nodes, respectively, with regard to A_j.

We can now derive the logic for assuring temporal validity, based on the following three propositions:

1. $P_1 \leftarrow A_i.canPlan \wedge \neg a_{i,j}.deadlock \wedge \neg p_{j,i}.deadlock \wedge \neg(a_{i,j}.solvable \wedge p_{j,i}.solvable)$
2. $P_2 \leftarrow A_j.canPlan \wedge \neg a_{j,i}.deadlock \wedge \neg p_{i,j}.deadlock \wedge \neg(a_{j,i}.solvable \wedge p_{i,j}.solvable)$
3. $P_3 \leftarrow A_i.canPlan \wedge A_j.canPlan \wedge \neg b_{i,j}.deadlock \wedge \neg b_{j,i}.deadlock \wedge \neg(b_{i,j}.solvable \wedge b_{j,i}.solvable)$

All three propositions essentially represent the same logic: we may plan our subgoal if we are currently in a safe node, it does not result in a deadlock state for us, it does not result in a deadlock state for another agent, and it does not result in both of us being in solvable states. Recall that a solvable state indicates we are waiting on another agent, so if we are waiting on one another this represents classic deadlock.

If P_3 holds then the agents may plan concurrently. If $(P_1 \wedge P_2)$ holds, but not P_3, this indicates that one agent may plan while the other must wait, regardless which one plans. Last, if P_1 holds but P_2 does not, then agent A_i may safely plan. If this logic holds across all agents, then it is safe for all agents to plan the next subgoal. Once a subgoal has been planned for by the rest of the architecture, the agents may attempt to enter into coordination again for the next subgoal.

3.1.3.2 Conflict Minimization

For the second approach, instead of agents taking a greedy approach to resolving temporal consistency, we take a conservative approach to minimizing conflicts. It is similar to Clement's Fewest Threats First heuristic, found in [3].

As with the previous algorithm, the agents still exchange their active, passive and both nodes. However, in this approach we strive to minimize the conflicts seen in the local node information. This is similar to the work done in [1]. More specifically, for a given agent we look at how badly we interfere with all other agents, in addition to how badly our current plan affects us. We achieve this by having each agent enter into an auctioning process with all other agents. Assuming we have n agents, we quantify this for each agent A_i against every other agent A_j using the following bid computation:

```
**if** $\bigvee_{j=0...n}(p_{j,i}.deadlock \vee a_{i,j}.deadlock \vee (a_{i,j}.solvable \wedge p_{j,i}.solvable))$ **then**
    $bid_i = \infty$
**else**
    $bid_i = \sum(p_{j,i}.solvableCost + a_{i,j}.solvableCost)$
**end if**
```

If we take an action (e.g., moving in the Sokoban example) that results in deadlock with respect to all other agents, or we cause another agent to go to deadlock, then our bid is infinity—we are unable to achieve our subgoal without resulting in deadlock. If this is not the case, then our bid is based on how long (in terms of qualitative time) all other agents must wait as a result of our subgoal being planned, as well as how much we may have to wait. The agents with the smallest bids (including ties) may plan ahead for their subgoals.

One major difference between this and the previous algorithm is that it enforces strict synchronization as to when subgoals may be considered valid. Subgoals that win the auction are then processed by the rest of the architecture—however note that we must wait for all valid subgoals to complete being planned for before we determine valid subgoals again. The end result is that the auction process produces all valid subgoals up through the completion of their planning—we may not enter into coordination for subsequent subgoals until *all* subgoals determined by the auction have completed.

3.2 Contracting Subgoals

In Section 3.1 we described the mechanism by which agents can efficiently identify conflicts that they may have with other agents. If we assume that each agent has found at most one subgoal to be temporally consistent, then the problem reduces to which agent should be responsible for producing a plan for each subgoal. In this section we present how the agents can determine this using an auctioning mechanism.

3.2.1 Auctioning Process

Assume agent A_i has a subgoal G that has been found to be temporally valid. Auctioning the entire subgoal is unwise, as it places a burden on the winning agent to solely complete the task, as well as making it difficult to determine a wise bid to place on such an abstract task. Instead, we will auction all leaf subgoals of G (assuming G is not itself a leaf subgoal), defined by $G = (g_1, g_2, ..., g_n)$, one at a time. Subgoals that are auctioned off are removed from G; subgoals where no agent places a valid bid remain in G for the next auctioning procedure, and no further subgoals of G are processed (recall Figure 2.1).

As agents win subgoals, they add them to a queue of pending subgoals S. Subgoals are then processed one at a time, much like they were auctioned off one at a time: to maintain temporal consistency. Temporal consistency is not guaranteed in this manner however, as a single agent may contain a subgoal belonging to a different agent whose subgoal at the previous qualitative time has not yet been completed. Hence, an agent may only map out plans for a subgoal if it is found to be temporally consistent: the qualitative time for that subgoal is met. To achieve this, each subgoal contains a unique identifier of the agent it originally belonged to. When a subgoal has been fully planned for, that agent tells all other agents of this event, therefore agents always know the current qualitative times of all subgoals.

It may seem unwise for an agent to auction off a subgoal which is dependent on the completion of a prior subgoal. There is a significant advantage in doing so, however. When an agent receives the subgoal, it may prepare for its completion while another agent (or agents) is completing the prior subgoal (or subgoals). Additionally, the auctioning mechanism is set up such that if no one chooses to complete a subgoal, it is held over into the next auctioning round. An agent may choose to do so in such situations where it is overwhelmed by the amount of pending auctions it must accomplish, and is unable to decide on another subgoal.

3.2.2 Bid Determination

An agent's bid for a subgoal is primarily dependent on the agent's current status (for instance, its possession of resources), as well as its pending auctions queue S. A bid effectively represents how much it will cost for an agent to complete a subgoal. Of course, this cost may only be approximated as it is impossible to predict the environment at the qualitative time of the particular subgoal.

Costs, where cost is defined as the number of primitive actions necessary to complete a subgoal, are maintained for every subgoal of the pending auctions queue. If the queue is empty, then the cost it takes to achieve a subgoal will be rather reasonable. However, as our queue increases, the cost estimates become far more inaccurate. We therefore define two aspects to estimating cost: *optimistic approximation* and a *user-defined dampening function*. The optimistic approximation a is the cost it takes for a plan to complete a subgoal, assuming the environment is free of all other agent resources. The dampening function d takes in the size of the pending auctions queue and returns a value designed to dampen the optimistic approximation. We define the full bid as a recurrence relation, for subgoal g:

$$\begin{aligned} bid_0(g) &= d(0) * a(g) \\ bid_i(g) &= bid_{i-1}(g) + d(i) * a(g) \end{aligned}$$

In bid_i, i is the size of the pending auctions queue. We note that d should be a nondecreasing function, since as the size of S increases our approximations grow worse. Additionally if the size of S is so large that it becomes unreasonable to bid on an auction, we may set a threshold on d such that at some large value of the size of S it returns infinity.

In the context of Sokoban this form of bidding works particularly well. For optimistic approximation, in situations where S is empty we apply breadth-first search in finding the shortest path to the box. In searching the space we take into account all boxes in the scene, but ignore all movers. The assumption made is that a mover's position should not be accounted for since we will resolve mover-to-mover conflicts via resource reservation (discussed in the next section). If S contains goals, we simply use Manhattan distance as a way of computing the smallest distance it takes for a mover to push a box. The dampening function is defined as $d(x) = (x + 1)^2$, defined for queue sizes up to 4. Beyond this it returns infinity.

3.2.3 Discussion

Once agents are assigned subgoals to achieve, they map out their plans as defined in the next section. However, note that agents must all agree on the synchronization policy defined by the first level of the architecture. Coordination at the first level implies that all subgoals divided among agents may be planned for, but *nothing* beyond this. Thus agents must wait for all qualitative time to be in the state agreed upon in the first-level coordination before proceeding, where qualitative time essentially acts as barrier synchronization for agents.

The algorithm is similar to the work done in [10], in that an agent may allow for other agents to complete subgoals for it, effectively repairing its plan. However, our system also handles two important properties. Effects may be irreversible, as we have resolved subgoal conflicts at the first level of our architecture. Additionally we allow for resources to be in contention among multiple agents, which is detailed in the third level of our architecture.

There are a couple of advantages to using an auctioning process for subgoal completion, as opposed to relying on an agent's original intentions. In dynamic environments, the once valid plan for completing an agent's subgoal can quickly become invalid or difficult to accomplish. If it takes another agent significantly fewer resources to achieve the subgoal, then we should let them handle it. Secondly, subgoal auctioning tends to localize agents to resources in the environment. The subgoals they take on will likely be close to the resources they are currently using, therefore agents will localize themselves to parts of the environment, reducing the chance of resource conflicts.

3.3 Resource Reservation

After agents obtain subgoals from the auction, they may not simply refine them and add the actions to their plan, as in [10]. Despite agents containing separate subgoals to achieve, it is very possible for them to run into resource conflicts as their primitive plans are fleshed out. Thus we must allow for a mechanism in which an agent's plan for a subgoal is guaranteed to be conflict free. We handle this via resource reservation.

Assume that an agent A_i has an ordered set of subgoals to achieve Q_i. Also assume that $q \in Q_i$ is the first subgoal to achieve. Subgoal q must be such that an agent is able to produce a coarse representation of the resources required to achieve it. Otherwise, it is necessary to expand on q's children in the subgoal hierarchy until we are able to do so.

Once A_i has represented its resource usage, it broadcasts the *zone request*, $Z = \langle r_i, c_i, p_i, g_i \rangle$, to all other agents, where r_i represents resource usage, c_i is the lower bound to completing the subgoal, p_i indicates the agent priority in completing the subgoal, and g_i is the subgoal to be completed. We will discuss priority in the next section. As soon as an agent collects all of the other agent zones, it computes resource overlap with all other zones, and determines the smallest lower bound cost among all agents.

With this in mind, an agent may then plan for a subgoal, following two rules: we may not use resources in zones whose priority is greater than ours, and the length of our plan must not exceed the smallest lower bound cost. Refer to Figure 3.5 for the zoning algorithm. If no zones with higher priority overlap an agent's zone, then the agent may plan under the knowledge that all resources

```
function zone (subgoal queue $Q$, current plan $P$)
while $Q$ contains subgoals do
   $q \leftarrow top(Q)$
   $r \leftarrow resourceUsage(q)$
   $p \leftarrow computePriority(q, Q, P, r)$
   $c \leftarrow lowerBound(q, P)$
   broadcast $\langle r_i, c_i, p_i, g_i \rangle$
   $\forall_j$ collect $Z_j = \langle r_j, c_j, p_j, g_j \rangle$
   $R \leftarrow replan(q, Z_j)$
   append all $r \in R$ to $P$
   if q.isCompleted then
      pop($Q$)
   end if
end while
```

Figure 3.5: Zone coordination algorithm.

currently in the environment (namely resources that other agents control) will remain static. If a set of zones with higher priorities completely takes over our zone, then we must wait for these zones to finish, or for these agents to subdivide their plans and refine their zones. If a set of zones with higher priorities only partially overlaps our zone, then we may plan under the knowledge that these resource zones are inaccessible to us.

3.3.1 Zone Priority and Lower Bound Cost

Zone priority effectively represents our bid for planning our subgoal. It is thus a function of how costly our plan is (the length of the plan) and our intention (achieving a subgoal or not). Note that agents who only require a small number of resources to satisfy their subgoal will likely finish their subgoal in a small amount of time. Thus, priority is derived by an estimate of how long an agent's plan will be for completing a subgoal. A smaller amount of resource usage will result in a higher priority.

For Sokoban, computing the lower bound to completing a subgoal is straightforward: we again use Manhattan distance, as it represents the smallest distance in moving to a box. Computing priority via the actual amount of resources an agent will use is, as already discussed, difficult to determine. We use the same technique as discussed in Section 3.2.2 where we find shortest-path distance, except we take into account both movers and box positions. In planning for subgoals we set the priority to be the cost. When we are not planning for a subgoal (for instance, anticipating a subgoal) we set the priority to be twice the cost. Subgoal completion is our main priority.

In applying this to a variety of scenarios in Sokoban, we find that although resource reservation is rather course-grained, agents still exhibit a high level of concurrency in planning for subgoals. In the case of smaller zones embedded in larger zones, the agent containing the smaller zone will have the higher priority and thus may plan free of conflict, while the agent containing the larger zone may plan around the smaller zone. Therefore, plans remain consistent, and the need to replan and/or plan-repair is diminished.

Zones may also be used in resolving conflicts where subgoals become unattainable due to the rapidly changing environment. An agent A_i may broadcast its zone $Z_i = \langle r_i, \infty, \infty, g_i \rangle$ to indicate that it is unable to achieve its subgoal. This has two implications on the other agents. First, an agent which is found to be in conflict with g_i, but has a subgoal to complete, may find that the completion of its subgoal results in no longer being in conflict with A_i. The original algorithm inherently resolves this situation. The second case deals with an agent which may not have a subgoal to achieve, but is in possession of resources that result in A_i not being able to achieve its subgoal. In this case, the agent may construct a new plan to free up resources.

4 Results and Discussion

We have implemented our 3-level multi-agent planning architecture in a set of agents solving the multi-agent Sokoban problem. As discussed in Chapter 3, our implementation of a Sokoban game server is based the Asynchronous Chess server [8], modified for the Sokoban domain to support synchronized actions. The running example (Figure 4.1, repeated from Figure 2.1) demonstrates the 2-agent case. Recall that the pawns represent the respective movers which belong to each agent. The white pawn belongs to agent A and the black pawn belongs to agent B.

Figure 4.1a show the resulting plans that each agent generates for achieving its assigned goals (e.g., boxes to be moved to goal locations). The stars ($\star$) represent the goal locations for each agent, and each agent is assigned a set of boxes (e.g., the white agent plans for the white box to be pushed into the white goal). In constructing the subgoal hierarchy, we view a single box push as being the lowest level subgoal, corresponding to a primitive "movement" action. More abstract goals are represented as being a series of box pushes, or in terms of resources, the full amount of area in which boxes are pushed. For instance, agent A's second level hierarchy depth indicates that each subgoal is essentially the action of pushing a box 3 times. Similarly, agent B has only 3 subgoals in its plan, corresponding to moving the box up one position, and then left two positions. Figure 4.1c shows the "resources" (i.e., board positions) that each agent requires to accomplish its plan.

Once the agents have generated their individual plans, they coordinate their actions in order to avoid conflict. As an example, consider the deadlock graphs for the running example shown in Figure 4.2, derived from the temporal constraint matrices in Section 3.1.1 as described in by the $constructDeadlockGraph$ algorithm in Figure 3.3. Using the coordination algorithm (Figure 3.4), it is found that the second level of the subgoal hierarchies for both agents is adequate in determining temporal validity. Using either of the algorithms (Sections 3.1.3.1 and 3.1.3.2) for determining temporal validity will in fact give us a solution.

For the *temporal validity* algorithm outlined in Section 3.1.3.1, no form of synchronization is enforced, yet subgoals still will not conflict in spite of this. For instance, agent A may completely plan for its subgoals while agent B simply does nothing. Note that this results in B being placed in the last column of its deadlock graph, where it may safely plan ahead. On the other hand, if B completely plans for its subgoals before A does, we find that in A's last column of the graph we will encounter deadlock. Thus, B may only plan for two of its subgoals before waiting for A to proceed.

The *conflict minimization* algorithm outlined in Section 3.1.3.2 enforces a much stricter policy on synchronization. The auctioning process will result in the following ordering for subgoals being

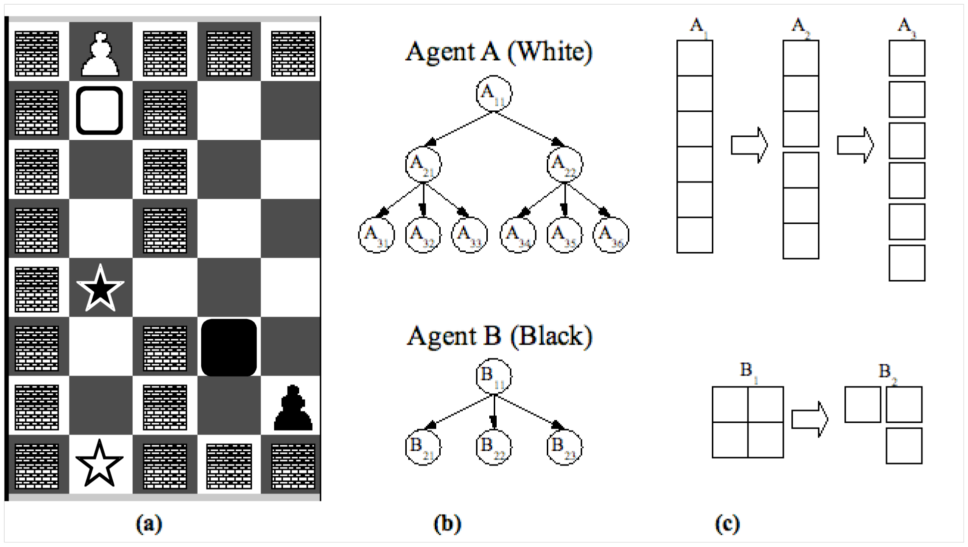

Figure 4.1: An example Sokoban scenario. (a) The white agent must push the white box all the way down, while the black agent must push the box up, then push it to the left. (b) The resulting subgoal hierarchies for both agents. (c) Resource usage for all subgoals.

planned for:

- qualitative time 1: $A_{2,1}, B_{2,1}$ (concurrently planned)
- qualitative time 2: $A_{2,2}$ (A's goals completed)
- qualitative time 3: $B_{2,2}$
- qualitative time 3: $B_{2,3}$(B's goals completed—global goal completed)

Note that at qualitative time 2, only agent A may allow for its subgoal to be planned. Agent B has the possibility of entering a solvable state if its goal is planned for, so it loses the auction.

The method of assuring temporal validity is effective for environments in which only two agents are planning. When more than two agents are planning, this algorithm does a poor job at resolving the more intricate conflicts that arise between multiple agents. It is a rather relaxed means of coordination, in that while it may allow for larger concurrency among subgoals being temporally consistent, it does not quantify the effects of traversing nodes in the deadlock graphs. On the other hand, it is an efficient means of subgoal coordination. The time complexity is $\mathbf{O}(a)$ in the number of agents to communicate.

Conflict minimization, while requiring twice the amount of communication than that of the other algorithm (but still linear in the number of agents), enforces tighter synchronization among the ordering in which subgoals are considered temporally valid. The motivation behind this is for an agent with the smallest amount of interference to allow for planning, in the hopes that the

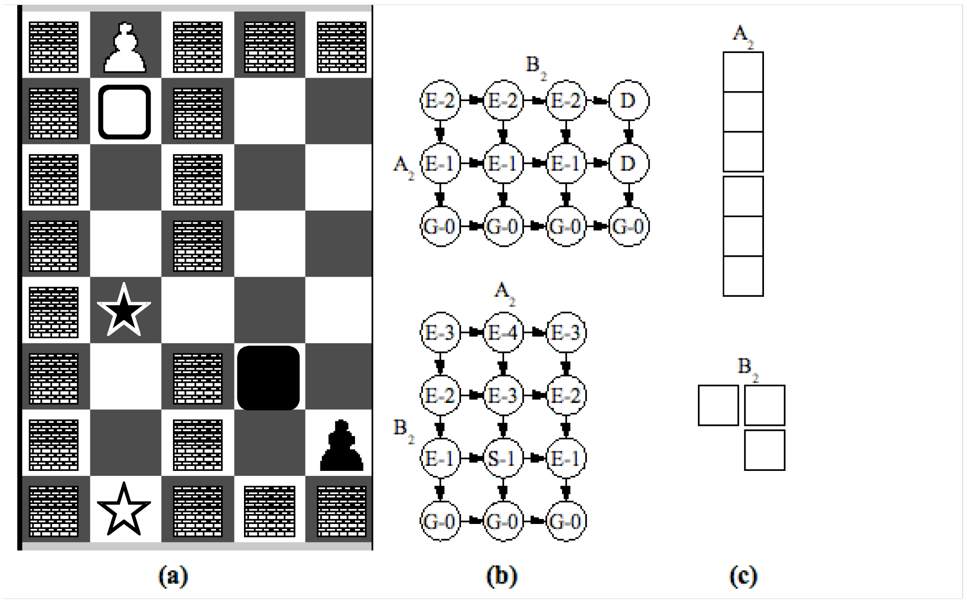

Figure 4.2: (a) The running Sokoban example. (b) Associated deadlock graphs for each agent. "E" nodes are safe nodes, "S" nodes are solvable nodes, "D" nodes are deadlock, and "G' nodes are goal nodes. (c) Associated resource usage for both agents.

other agents will subsequently traverse their deadlock graphs to better states. In the context of Sokoban, we find that the amount of concurrency is reduced, but more intricate agent conflicts become resolved through this auctioning method.

4.1 Limitations and Future Work

The focus of the system architecture described in this paper is on maintaining plan consistency, but nothing beyond this. Using the terminology of [4] subgoal hierarchies strictly consist of *and* subgoals and no *or* subgoals. In other words we do not allow any subgoal alternatives. Incorporating *or* plans into the current architecture will lead to more expensive search for subgoal coordination (refer to Section 3.1.3) and a larger number of deadlock graphs.

Additionally our system only handles a certain subset of problems. It is unable to resolve conflicts for which replanning/plan-repair is required to handle, such as in the Sokoban example shown in Figure 4.3. While replanning for this particular example is rather simple, as described, for more dynamic environments it becomes unmanageable. Having *or* plans incorporated in the subgoal hierarchy may still not resolve the more complicated environments.

We believe that deadlock graphs, however, may be used in guiding plan-repair. In using plan-repair, plan modification reduces to modifying the deadlock graph structure such that no form of inevitable deadlock exists. Since deadlock graphs contain very useful information with regard to what leads to deadlock at what qualitative times, we believe that they could be beneficial for

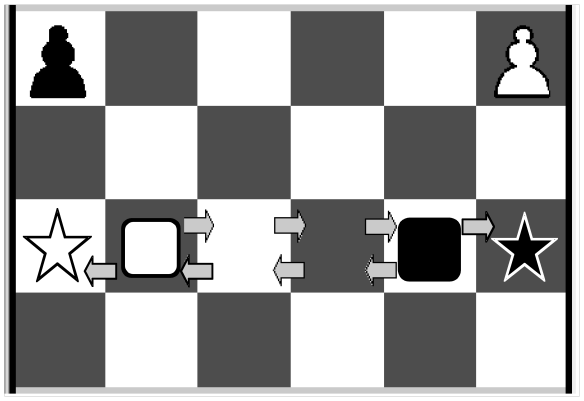

Figure 4.3: Sokoban example that current system is unable to resolve.

informed planning. Rigid coordination must be used in such situations, so that agents may agree on who should modify their plans. In doing so, every other agent may safely adjust its deadlock graphs.

5 Conclusion

We have presented a system architecture for handling multi-agent planning in highly dynamic environments. Our work focuses on maintaining plan consistency based on the observation that replanning/plan-repair in such environments becomes difficult to manage. We have developed and implemented a 3-level system architecture based on each level resolving fundamental agent conflict issues: *when* is it valid to plan, *who* should plan, and *how* should the plan be produced. The system is suited for environments in which a goal may be partitioned among agents in order to make the goal more manageable to complete.

6 References

[1] T. Bartold and E. Durfee. Limiting disruption in multiagent replanning. In *Proc. of the Second International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 49–56, 2003.

[2] G. Braun. AFFOR Command and Control Enabling Concept - Change 2. AF/A5XS Internal Report, 25 May 2006.

[3] B. Clement. *Abstract Reasoning for Multiagent Coordination and Planning*. PhD thesis, University of Michigan, Department of Electrical Engineering and Computer Science, 2002.

[4] B. Clement and Durfee. Top-down search for coordinating the hierarchical plans of multiple agents. In *Proc. of the Third Annual Conference on Autonomous Agents*, pages 252–259, 1999.

[5] J. Culberson. Sokoban is pspace-complete. Technical Report TR97-02, University of Alberta, 1996.

[6] K. Erol, J. Hendler, and D. Nau. Semantics for hierarchical task-network planning. Technical Report CS-TR-3239, University of Maryland, March 1994.

[7] M. Fox, A. Gerevini, D. Long, and I. Serina. Plan stability: Replanning versus plan repair. In *Proceedings of International Conference on AI Planning and Scheduling (ICAPS-06)*, 2006.

[8] N. Gemelli, R. Wright, and R. Mailer. Asynchronous chess. In *Proc. of the AAAI Fall Symposium on Co-Adaptive and Co-Evolutionary Systems*, 2005.

[9] T. Sugawara, S. Kurihara, T. Hirotsu, K. Fukuda, and T. Takada. Predicting possible conflicts in hierarchical planning for multiagent systems. In *Proc. of the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 813–820, 2005.

[10] R. van der Krogt and M. de Weerdt. Self-interested planning agents using plan repair. In *Proceedings of the ICAPS-05 Workshop on Multiagent Planning and Scheduling*, pages 36–44, 2005.

7 Symbols, Abbreviations and Acronyms

AF	Air Force
AFRL	Air Force Research Laboratory
AI	Artificial Intelligence